

**МИНОБРНАУКИ РОССИИ**  
**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**  
**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**  
**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**  
**Кафедра МО ЭВМ**

**ОТЧЕТ**  
**по курсовой работе по**  
**дисциплине «АиСД»**  
**Тема: Реализация идеально сбалансированного БДП**

Студент гр. 8304

\_\_\_\_\_

Бочаров Ф.Д.

Преподаватель

\_\_\_\_\_

Фирсов М. А.

Санкт-Петербург

2019

**ЗАДАНИЕ  
НА КУРСОВУЮ РАБОТУ**

Студент Бочаров Ф.Д.

Вариант 21

Группа 8304

Тема работы: Реализация идеально сбалансированного дерева.

Исходные данные:

Пользователю предоставляется возможность добавить элемент и удалить элемент. Построить бинарное дерево.

Предполагаемый объем пояснительной записки:

Не менее 14 страниц.

Дата сдачи реферата:

Дата защиты реферата:

Студент

\_\_\_\_\_

Бочаров Ф.Д.

Преподаватель

\_\_\_\_\_

Фирсов М.А.

## **АННОТАЦИЯ**

В ходе выполнения курсовой работы была разработана программа GUI, демонстрирующая построение идеально сбалансированное дерево. Программа добавляет элемент и балансирует при необходимости дерево. Также реализовано удаление элемента, с возможность повторения операции.

## **SUMMARY**

In the course of the course work, a GUI program was developed, demonstrating the construction of a perfectly balanced tree. The program adds an element and balances the tree if necessary. Also implemented is the removal of the element, with the possibility of repeating the operation.

## Оглавление

Введение .....	5
Описание алгоритма. ....	6
Структура данных. ....	7
Тестирование программы. ....	9
Список литературы. ....	11
Вывод .....	12
Приложение. ....	13

## **Введение**

Дерево – это совокупность элементов, называемых вершинами, или узлами, связанных между собой отношениями вида «родитель – сын». Отношения отображаются в виде линий, которые называются рёбрами, или ветвями дерева. Узел дерева, не имеющий предков, называется корнем дерева, а узлы, не имеющие потомков, называются листьями дерева.

Деревья обычно отображаются по уровням. На нулевом уровне находится корень дерева, на первом – его сыновья, на втором – сыновья этих сыновей и т.д.

Уровень каждого элемента называется также его глубиной, а количество уровней в дереве называется глубиной дерева. Дерево называется бинарным (двоичным), если каждый его узел имеет максимум двух сыновей.

Бинарное дерево называется идеально сбалансированным, если для каждого его узла количество узлов в левом и правом поддеревьях отличается максимум на единицу.

## **Цель работы**

Изучить бинарное дерево поиска и его реализацию на языке C++, а также сбалансировать его. Реализовать операцию вставки элемента, удаления и визуализировать дерево.

## Описание алгоритма

С помощью функции `insert` каждый элемент последовательно добавляется в бинарное дерево поиска таким образом, чтобы не нарушалось правило построения АВЛ-дерева (ключ любого узла дерева не меньше любого ключа в левом поддереве данного узла и не больше любого ключа в правом поддереве этого узла).

После вставки элемента дерево проверяется на сбалансированность (для любого узла дерева высота его правого поддерева отличается от высоты левого поддерева не более чем на единицу). Если дерево не сбалансированно, то вызывается функция `balance`, которая работает следующим образом:

Если разница высот правого и левого поддеревьев равна 2, выполняется правый поворот.

Если разница высот правого и левого поддеревьев равна -2, выполняется левый поворот.

Если высота правого поддерева на 2 больше высоты левого поддерева, выполняется либо простой поворот влево вокруг `p`, либо так называемый большой поворот влево вокруг того же узла.

После того как бинарное дерево построено, пользователю предлагается выбрать элемент и удалить его из дерева. Для этого вызывается функция `remove`, которая находит узел с заданным ключом (бинарный поиск) и `min` узел с наименьшим ключом в правом поддереве, а затем заменяет удаляемый узел `p` на найденный узел `min`. Если найденный узел не имеет правого поддерева, то по свойству АВЛ-дерева слева у этого узла может быть только один единственный дочерний узел (дерево высоты 1), либо узел вообще является листом. В обоих этих случаях функция удаляет узел и возвращает в качестве результата указатель на левый дочерний узел найденного узла.

## Структуры данных

```
struct node
{
    int key;
    unsigned char height;
    node* left;
    node* right;
    explicit node(int k) { key = k; left = right = nullptr; height = 1; }
}
```

Структура узла дерева

Key – значение узла

height – высота дерева

left, right – указатели на левое и правое дерево

node – конструктор для структуры node

```
unsigned char height(node* p)
```

Метод для определения высоты дерева.

```
int bfactor(node* p)
```

Фактор балансировки(разность между высотой правого и левого дерева; не должен превышать 1 по модулю). В процессе добавления или удаления узлов в AVL-дерево возможно возникновение ситуации, когда balance factor некоторых узлов оказывается равными 2 или -2, т.е. возникает *расбалансировка* поддерева.

```
void fixheight(node* p)
```

Метод для подсчета высоты.

```
node* rotateright(node* p)  
node* rotateleft(node* q)
```

Методы для левого и правого поворота.

```
node* insert(node* p, int k)
```

Метод для вставки ключа k в дерево p.

Вставка нового ключа в AVL-дерево выполняется так: спускаемся вниз по дереву, выбирая правое или левое направление движения в зависимости от результата сравнения ключа в текущем узле и вставляемого ключа. Строго доказывается, что возникающий при такой вставке дисбаланс в любом узле по пути движения не превышает двух, а значит применение вышеописанной функции балансировки является корректным.

```
node* remove(node* p, int k)
```

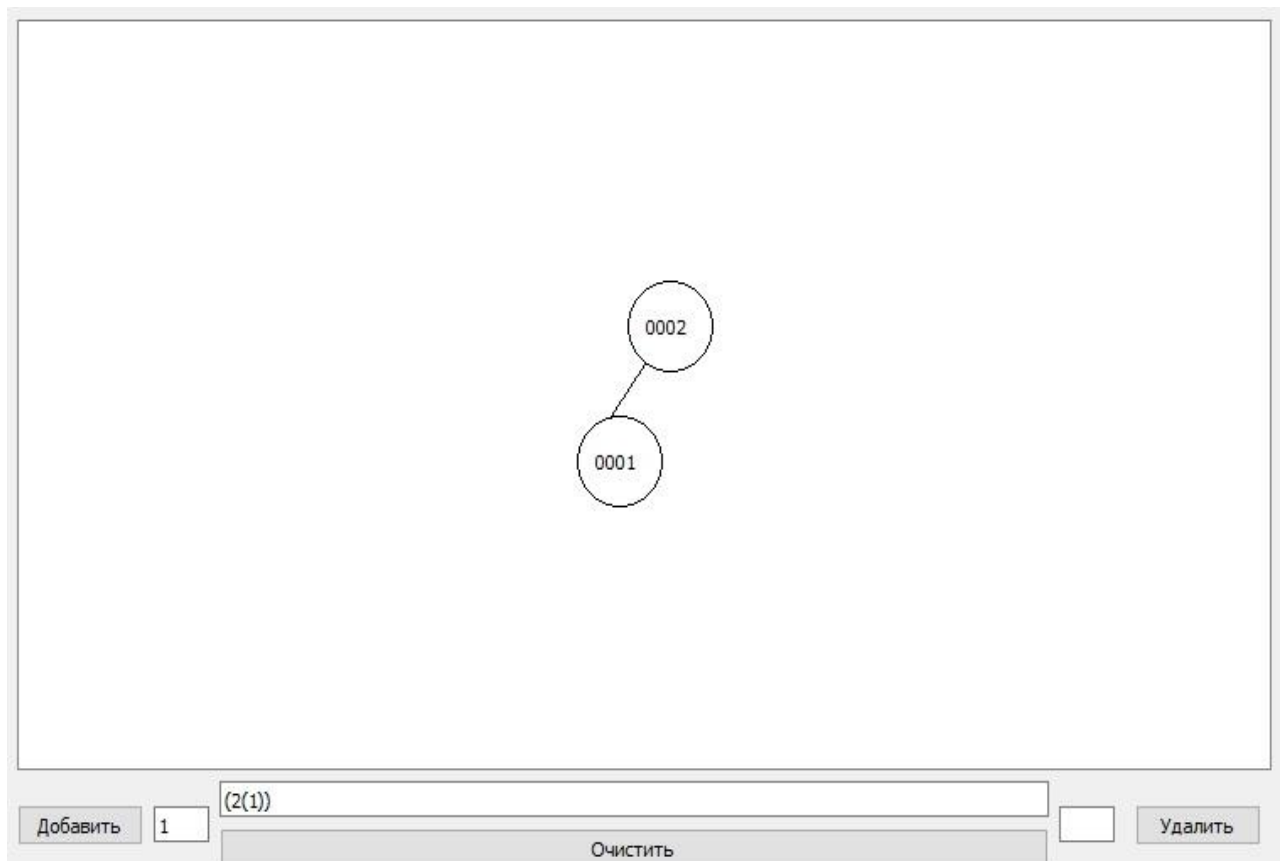
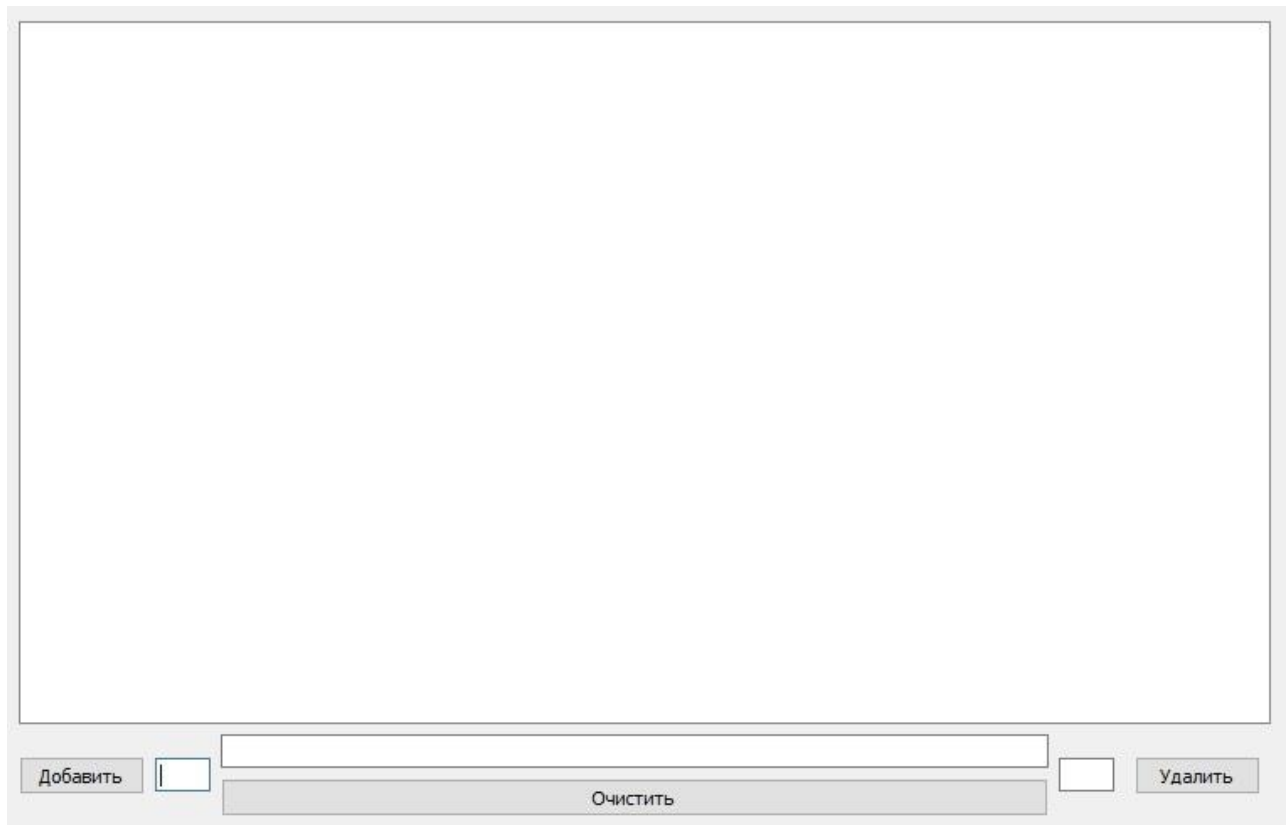
Метод для удаления ключа k из дерева p.

Находим узел p с заданным ключом k (если не находим, то делать ничего не надо), в правом поддереве находим узел min с наименьшим ключом и заменяем удаляемый узел p на найденный узел min.

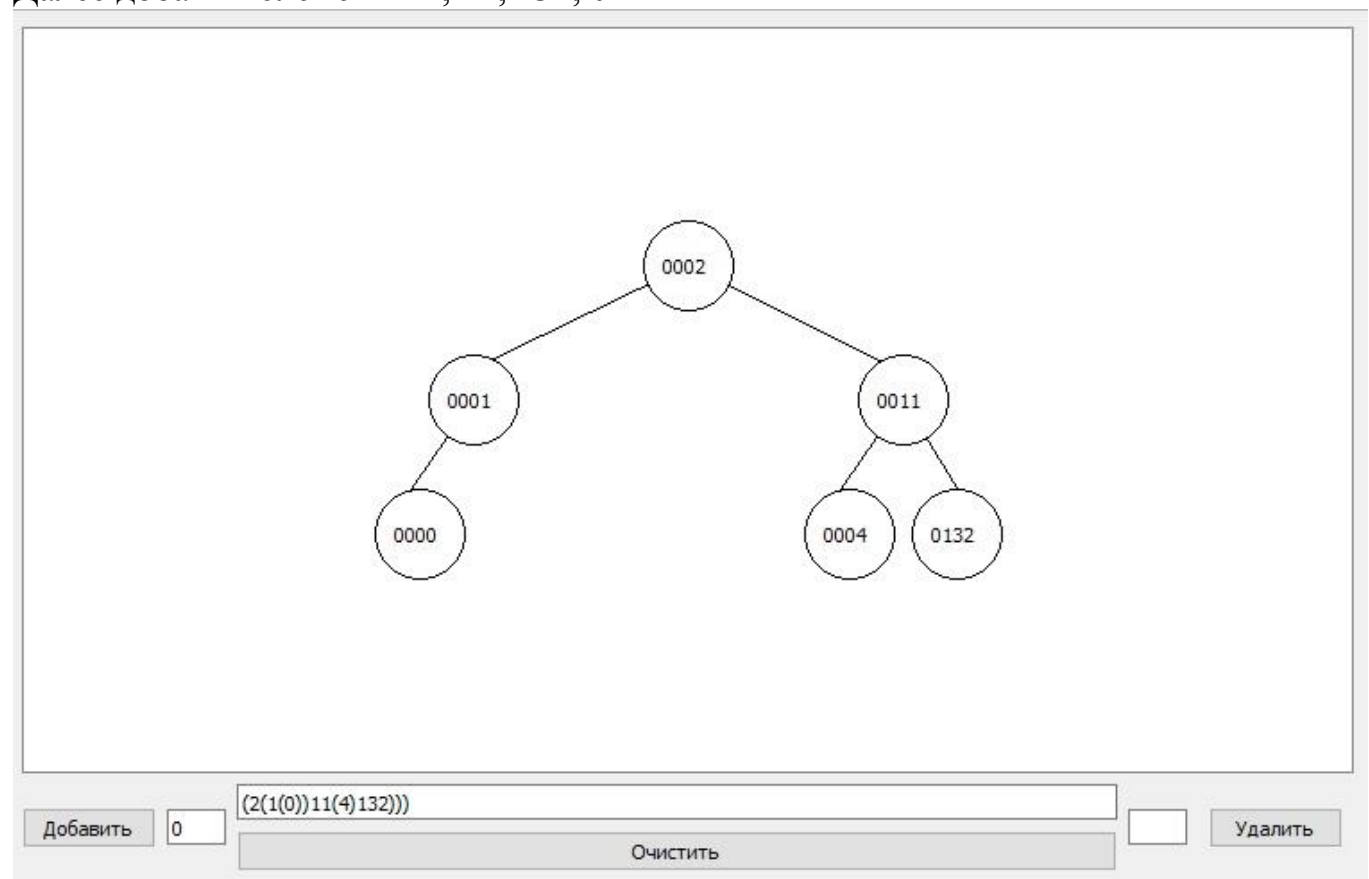


## Тестирование программы.

Изначально дерево пустое, добавим элементы 1 2



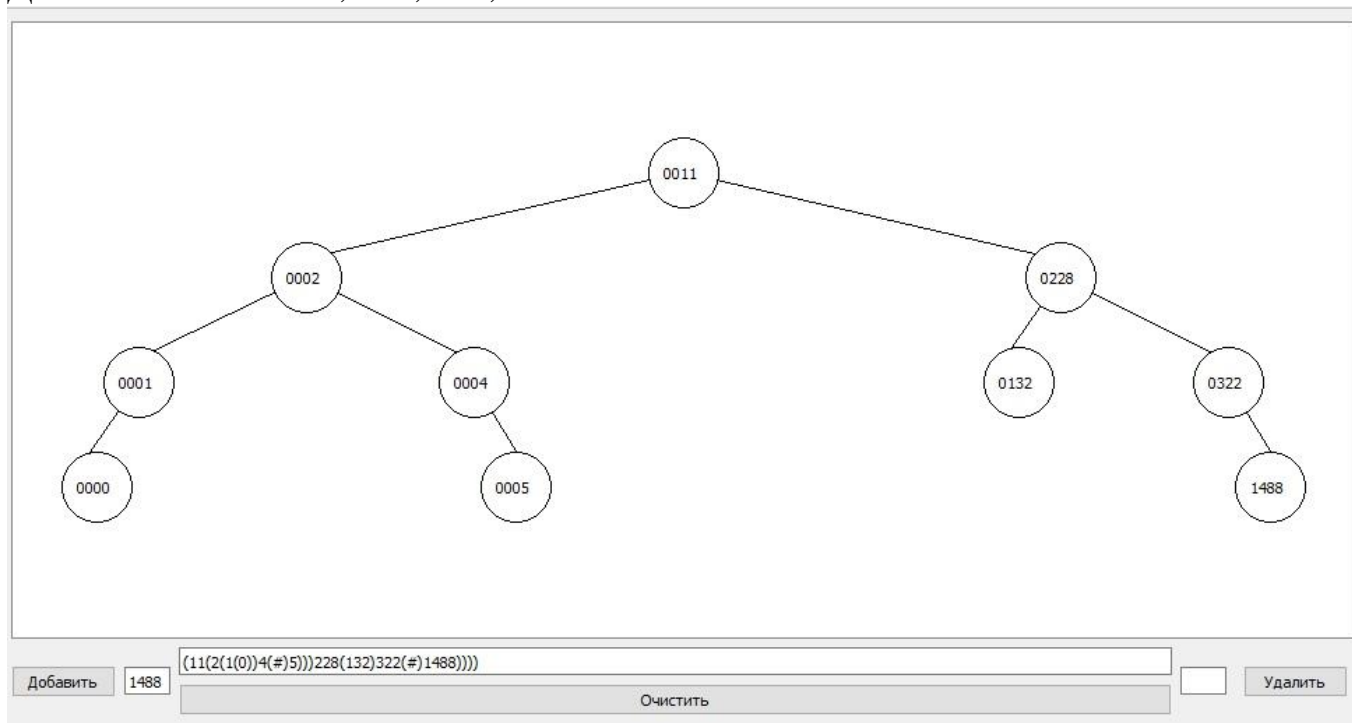
Далее добавим элементы 4, 11, 132, 0



```
insert element
insert element
insert element
insert element
insert element
balance factor 2, right rotate
insert element
```

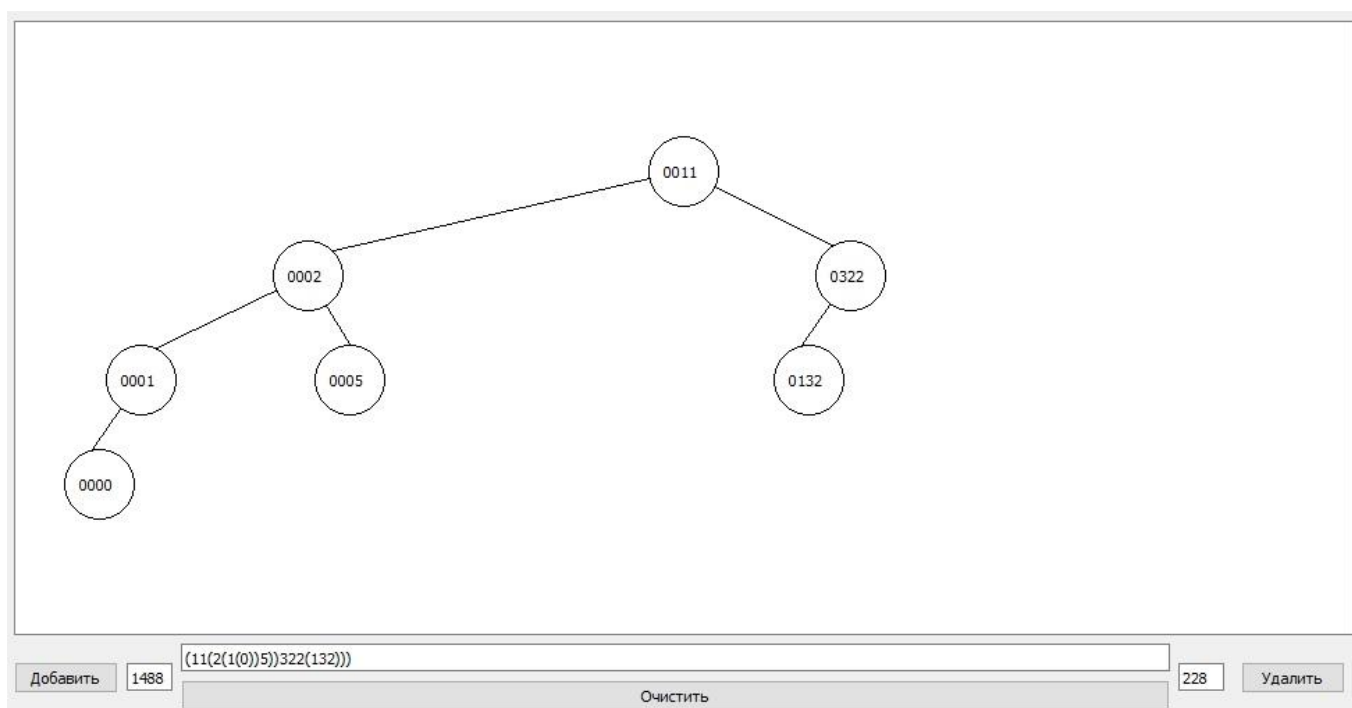
Баланс фактор равен двум, производится поворот чтобы сбалансировать дерево

Добавим элементы 5, 228, 322, 1488

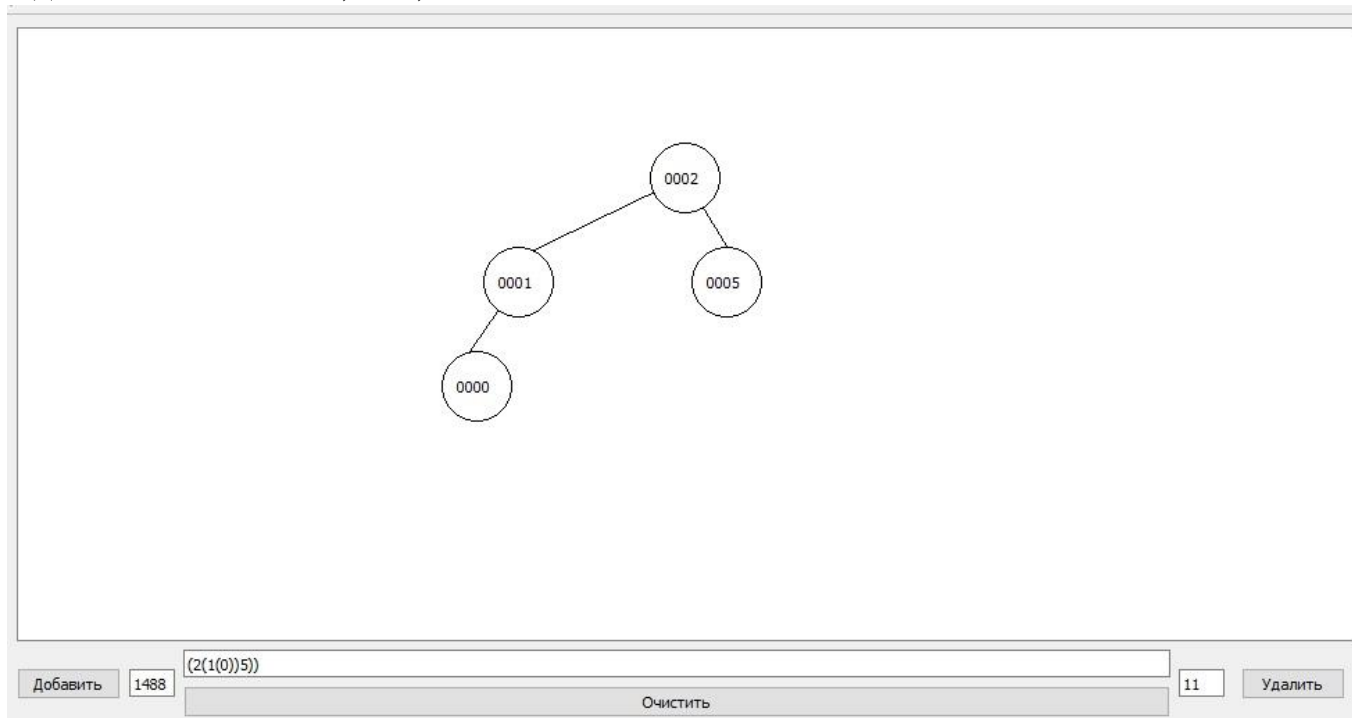


```
insert element
insert element
insert element
insert element
balance factor 2, right rotate
insert element
balance factor 2, right rotate
```

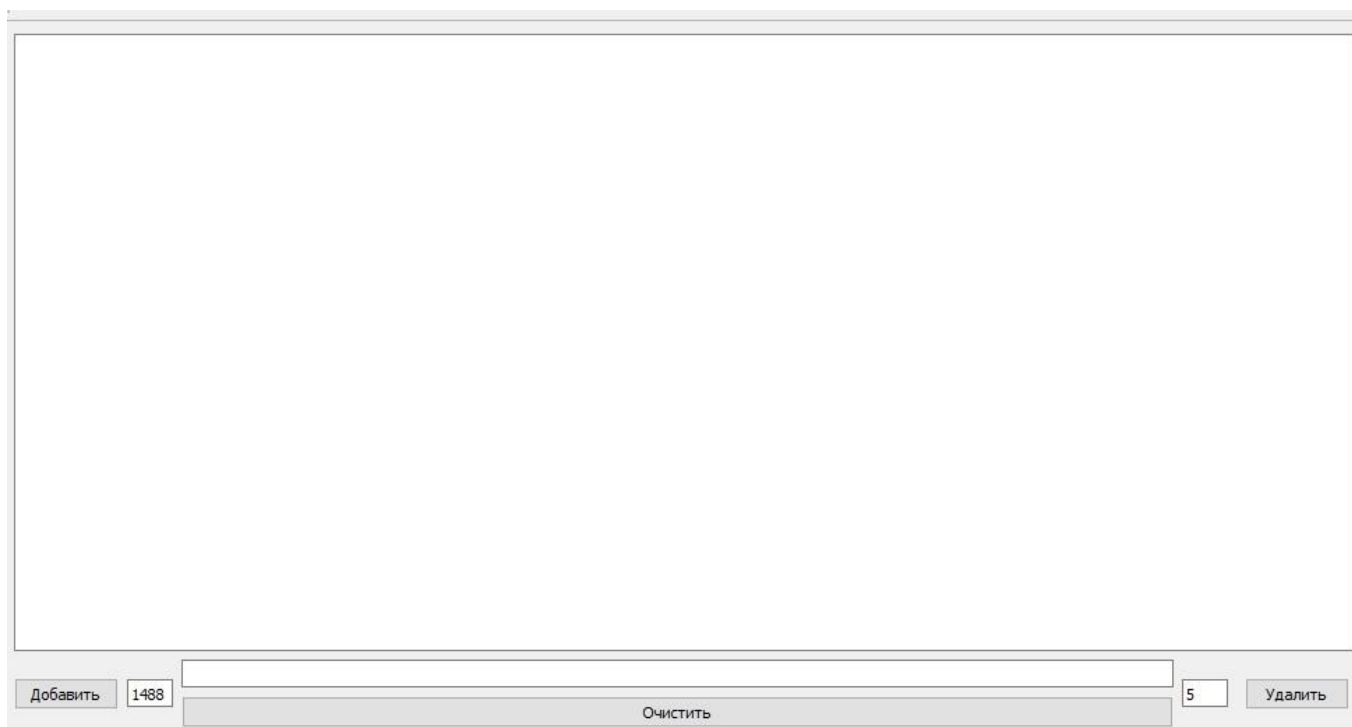
Потребовалось два поворота, чтобы сбалансировать дерево  
Теперь удалим элементы 4, 228, 1488



Удалим элементы 11, 322, 132



Удалим оставшиеся элементы



## Список использованных источников

1. Bjarne Stroustrup. A Tour of C++. М.: Addison-Wesley, 2018. 217 с.
2. Treap // GeeksforGeeks <https://www.geeksforgeeks.org/treap-a-randomized-binary-search-tree/> (дата обращения: 18.12.2000)
3. Qt Documentation // Qt. URL: <https://doc.qt.io/qt-5/index.html> (дата обращения: 18.12.2000)
4. Рандомизированные деревья поиска URL: <https://habr.com/ru/post/145388/#reed> (дата обращения: 17.12.2000)
5. The Height of a Random Binary Search Tree // BRUCE REED URL: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.152.1289&rep=rep1&type=pdf> (дата обращения: 18.12.2000)

## **Вывод**

В ходе работы было реализовано сбалансированное БДП со всеми основными функциями, добавление и удаление элемента, а так же балансировка дерева. Также была реализована визуализация при помощи средств фреймворка Qt.

## ПРИЛОЖЕНИЕ А

### Исходный код программы

#### Avl\_tree.h

```
#include
"avl_tree.h"

unsigned char height(node* p)
{
    return p ? p->height : 0;
}

int bfactor(node* p)
{
    return height(p->right) - height(p->left);
}

void fixheight(node* p)
{
    unsigned char hl = height(p->left);
    unsigned char hr = height(p->right);
    p->height = (hl > hr ? hl : hr) + 1;
}

node* rotateright(node* p) // правый поворот вокруг p
{
    node* q = p->left;
    p->left = q->right;
    q->right = p;
    fixheight(p);
    fixheight(q);
    return q;
}
```

```

node* rotateleft(node* q) // левый поворот вокруг q
{
    node* p = q->right;
    q->right = p->left;
    p->left = q;
    fixheight(q);
    fixheight(p);
    return p;
}

```

```

node* balance(node* p) // балансировка узла p
{
    fixheight(p);
    if( bfactor(p) == 2 )
    {
        if( bfactor(p->right) < 0 )
            p->right = rotateright(p->right);
        return rotateleft(p);
    }
    if( bfactor(p) == -2 )
    {
        if( bfactor(p->left) > 0 )
            p->left = rotateleft(p->left);
        return rotateright(p);
    }
    return p; // балансировка не нужна
}

```

```

node* insert(node* p, int k) // вставка ключа k в дерево с корнем p
{
    if( !p ) return new node(k);
    if( k < p->key )
        p->left = insert(p->left, k);
    else
        p->right = insert(p->right, k);
    return balance(p);
}

```



```

node* findmin(node* p) // поиск узла с минимальным ключом в дереве p
{
    return p->left ? findmin(p->left) : p;
}

```

```

node* removemin(node* p) // удаление узла с минимальным ключом из дерева p
{
    if( p->left == nullptr )
        return p->right;
    p->left = removemin(p->left);
    return balance(p);
}

```

```

node* remove(node* p, int k) // удаление ключа k из дерева p
{
    if( !p ) return nullptr;
    if( k < p->key )
        p->left = remove(p->left,k);
    else if( k > p->key )
        p->right = remove(p->right,k);
    else // k == p->key
    {
        node* q = p->left;
        node* r = p->right;
        delete p;
        if( !r ) return q;
        node* min = findmin(r);
        min->right = removemin(r);
        min->left = q;
        return balance(min);
    }
    return balance(p);
}

```

```

node* delete_tree(node* tree)
{
    if (!tree) return nullptr;

```

```

delete_tree(tree->left);
delete_tree(tree->right);

delete tree;

return nullptr;
}

void txt_tree(node* tree, string& str)
{
    if (!tree)
    {
        str = "";
        return;
    }
    stringstream sstream;
    string tmp;

    sstream << tree->key;
    sstream >> tmp;

    str += tmp;

    if (tree->left)
    {
        str += "(";

        txt_tree(tree->left, str);
    } else if (tree->right)
    {
        str += "(#";
    }
    if (tree->right)
    {
        txt_tree(tree->right, str);
    }
    str += ")";
}

```

## Mainwindow.cpp

```
#include
"mainwindow.h"

#include "ui_mainwindow.h"

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    ui->line_insert->clear();
    ui->line_remove->clear();

    abt = new About;
    scene = new QGraphicsScene(this);
    ui->visualization->setScene(scene);
}

MainWindow::~MainWindow()
{
    delete ui;
    delete scene;
    delete abt;
}

void MainWindow::on_button_insert_clicked()
{
    scene->clear();
    head = insert(head, int(stoi(ui->line_insert->text().toStdString())));

    print_tree(head, ui->visualization->geometry().x() / 2, 25);
    str_tree.clear();
    str_tree = "(";
    txt_tree(head, str_tree);
}
```

```

        ui->avl_tree->setText(QString::fromStdString(str_tree));
    }

void MainWindow::on_button_remove_clicked()
{
    scene->clear();
    head = remove(head, int(stoi(ui->line_remove->text().toStdString())));
    print_tree(head, ui->visualization->geometry().x() / 2, 25);

    str_tree.clear();
    str_tree = "(";
    txt_tree(head, str_tree);

    ui->avl_tree->setText(QString::fromStdString(str_tree));
}

void MainWindow::on_clear_clicked()
{
    foreach (QGraphicsItem* item, scene->items()) {
        delete item;
    }

    head = delete_tree(head);
    str_tree.clear();
    str_tree = "(";

    ui->line_insert->clear();
    ui->line_remove->clear();
    ui->avl_tree->clear();
}

```

```

void MainWindow::on_actionsave_triggered()
{
    QString path = QFileDialog::getOpenFileName(this, tr("Open path to save"),
"/home/egor/Desktop");

    if (path == nullptr) return;

    ofstream file(path.toStdString());

    file << (str_tree == "(" ? "(" : str_tree);
}

```

```

void MainWindow::on_actionopen_triggered()
{
    QString path = QFileDialog::getOpenFileName(this, tr("Open path to download
tree"), "/home/egor/Desktop");

    if (path == nullptr) return;

    ifstream oFile(path.toStdString());

    string str;
    oFile >> str;
    for (unsigned long i = 0; i < str.size(); i++)
    {
        if (str[i] == '(' or str[i] == ')' or str[i] == '#')
        {
            str[i] = ' ';
        }
    }
    stringstream sstream;
    int tmp;

```

```

sstream << str;

head = delete_tree(head);
str_tree.clear();
ui->avl_tree->clear();
ui->line_insert->clear();
ui->line_remove->clear();
str_tree = "(";

while(sstream >> tmp)
{
    head = insert(head, tmp);
}

txt_tree(head, str_tree);
ui->avl_tree->setText(QString::fromStdString(str_tree));

print_tree(head, ui->visualization->geometry().x() / 2, 25);

}

void MainWindow::on_actioninfo_triggered()
{
    (new HelpBrowser (":/docs/doc", "index.htm"))->show();
}

void MainWindow::on_actionabout_triggered()
{
    abt->show();
}

void MainWindow::print_tree(node* tree, int x, int y)
{
    if (!tree) return;

    const int offset = 30;
    const int r = 25;

```

```

        if (tree->left)
        {
            QLine line(x + r, y + r, x - offset * tree->left->height * tree->left-
>height + 10, y + 90);
            scene->addLine(line, QPen(Qt::black));
        }

        if (tree->right)
        {
            QLine line(x + r, y + r, x + offset * tree->right->height * tree->right-
>height + 35, y + 90);
            scene->addLine(line, QPen(Qt::black));
        }

        scene->addEllipse(x, y, 2 * r, 2 * r, QPen(Qt::black), QBrush(Qt::white));

        int temp = tree->key;

        int count_zero = 1;
        while (temp /= 10)
            count_zero++;

        QString zeroes = (tree->key >= 0 ? "" : "-");
        for (int i = 0; i < 4 - count_zero; i++)
            zeroes += '0';

        QGraphicsTextItem* txtItem =
            new QGraphicsTextItem(zeroes + QString::number(abs(tree->key)));

        if (tree->key >= 0)
            txtItem->setPos(x + 7, y + 15);
        else
            txtItem->setPos(x + 4, y + 15);

        scene->addItem(txtItem);

```

```
    if (tree->left) {  
        print_tree(tree->left, x - offset * tree->left->height * tree->left->  
>height, y + 75);  
    }  
  
    if (tree->right) {  
        print_tree(tree->right, x + offset * tree->right->height * tree->right->  
>height, y + 75);  
    }  
}
```